Analytics-driven Lossless Data Compression for Rapid In-situ Indexing, Storing, and Querying

Isha Arkatkar[1,2,7], John Jenkins[1,2,7], Sriram Lakshminarasimhan[1,2], Neil Shah[1,2], Eric R. Schendel[1,2], Stephane Ethier[3], CS Chang[3], Jackie Chen[4], Hemanth Kolla[4], Scott Klasky[2], Robert Ross[5], and Nagiza F. Samatova[1,2,6]

[1] North Carolina State University, NC 27695, USA
[2] Oak Ridge National Laboratory, TN 37831, USA
[3] Princeton Plasma Physics Laboratory, Princeton, NJ 08543, USA
[4] Sandia National Laboratory, Livermore, CA 94551, USA
[5] Argonne National Laboratory, Argonne, IL 60439, USA
[6] Corresponding author: samatova@csc.ncsu.edu
[7] Authors contributed equally

Abstract. The analysis of scientific simulations is highly data-intensive and is becoming an increasingly important challenge. Peta-scale data sets require the use of light-weight query-driven analysis methods, as opposed to heavy-weight schemes that optimize for speed at the expense of size. This paper is an attempt in the direction of query processing over losslessly compressed scientific data. We propose a co-designed double-precision compression and indexing methodology for range queries by performing unique-value-based binning on the most significant bytes of double precision data (sign, exponent, and most significant mantissa bits), and inverting the resulting metadata to produce an inverted index over a reduced data representation. Without the inverted index, our method matches or improves compression ratios over both general-purpose and floating-point compression utilities. The inverted index is light-weight, and the overall storage requirement for both reduced column and index is less than 135%, whereas existing DBMS technologies can require 200-400%. As a proof-of-concept, we evaluate univariate range queries that additionally return column values, a critical component of data analytics, against state-of-the-art bitmap indexing technology, showing multi-fold query performance improvements.

1 Introduction

Increasingly complex simulation models, capable of using high-end computing architectures, are being used to simulate dynamics of various scientific processes with a high degree of precision. However, coupled with this opportunity to augment knowledge and understanding of the highly complex processes being studied are the challenges of conducting exploratory data analysis and knowledge discovery. Specifically, data size on the tera- and peta-scale is becoming a limiting factor in understanding the phenomena latent in these datasets, especially in a post-processing context.

Due to massive dataset sizes, full context analysis is a crucial bottleneck in the knowledge discovery pipeline, being restrained by the limits of computer memory and I/O bandwidth. Most commonly, the applications that such data exploration processes

are characteristic of are interactive and require close to real-time I/O rates for full data exploration. However, I/O access rates are too slow to support efficient random disk access in real-time for large-scale data sets, necessitating new approaches geared towards reducing the I/O pressure of extreme-scale data analytics.

A *knowledge priors* approach to data analytics is promising in restricting data to smaller and more practical sizes. Often times, scientists have some prior knowledge about the regions of interest in their data. For example, fusion scientists aiming to understand plasma turbulence might formulate analyses questions involving correlations of turbulence intensities in different radial zones ($0.1 < \psi < 0.15; 0.3 < \psi < 0.35; 0.5 < \psi < 0.55; 0.7 < \psi < 0.75; 0.9 < \psi < 0.95$). Likewise, climate scientists aiming to understand factors contributing to natural disasters might limit their search to particular regions or perhaps only a single region.

Formulating queries on scientific simulation data constrained on variables of interest is an important way to select interesting or anomalous features from large-scale scientific datasets. Traditional database query semantics can effectively be used for formulating such queries. This allows us to leverage a great deal of work done in the database community on query processing. The indexing techniques used in traditional database systems, such as $B-$trees [11] or bitmap indexes [24], have been used extensively in the literature. However, while indexing is a blessing for fast and efficient query processing, it is arguably a curse in terms of storage; the index size is often 100-300% of the original column size for high-cardinality data (such as double-precision data), which is a huge bottleneck for storage-bound extreme-scale applications.

A number of bitmap index compression techniques have been introduced to reduce the size of the bitmap index while keeping fast query retrieval possible. In particular, Word Aligned Hybrid (WAH) [17] bitmap compression is used in FASTBIT [24], a state-of-the-art scientific database technology with fast query processing capabilities. Overall, the storage footprint used in FASTBIT for a high-cardinality column and its corresponding index is around 200% of the original size, which still becomes prohibitive for extreme-scale data sets. Furthermore, these indexing schemes are optimized for returning the record ID, or region index in the context of spatio-temporal data sets. However, for data analytics, the *actual values* of the variables associated with these points are equally important.

Therefore, we present a co-designed data reduction and indexing methodology for double-precision datasets, optimized for query-driven data analytics. We believe that a tight cohesion between the methods allows us to optimize storage requirements while at the same time facilitating both fast indexing at simulation-time and range query processing with value retrieval, desirable features for data analytics. Our focus in particular is on write-once, read-many (WORM) datasets utilizing double-precision floating-point variables, representing large-scale, high-fidelity simulation runs that are subsequently analyzed by numerous application scientists in multiple (often global) contexts. A few examples of such data are in the particle-based fusion simulation GTS [21] and in the direct numerical combustion simulation S3D [10], each of which are comprised of primarily double-precision, high-cardinality variables ($\approx$ 100% unique for GTS, $\approx$ 50% unique for S3D).

To be more specific, our paper makes the following contributions:

– We present a lossless compression methodology for floating-point (single and double-precision) columns that can be utilized for indexing and range query processing, utilizing unique-value encoding of the most significant bytes. Our lossless compression reduces the size of a number of high-entropy, double-precision scientific datasets by at least 15%. Compared to lossless compression techniques like FPC [8], optimized for double-precision data, we report superior average compression ratios.
– Using our lossless compression method, we optimize range query evaluation including value retrieval by binning the column data by the distinct significant byte metadata, integrating efficient compressed-data organization and decompression of retrieved results. Compared to state-of-the-art techniques like FASTBIT [24], we provide comparable or better performance on range queries retrieving record IDs. For range queries additionally retrieving variable values, we achieve a performance improvement by a factor of 28 to 38.
– For query processing, we utilize an inverted index that uses approximately 50% space with respect to the original column size. Considering both the compressed column data and index, our method has a smaller storage footprint compared to other database indexing schemes.

2 Background

Search and query processing operations on traditional database systems like Oracle [4], MySQL [3], and DB2 [5] involve the use of indexing techniques that are usually variants of either bitmap indexes or $B-$trees. While these techniques are effective in speeding up query response times, they come at the cost of a heavy-weight index management scheme. Indexing with $B-$trees [11], which tends to be more suitable for transactional databases that require frequent updates, is observed to consume storage that is three to four times the size of the raw column data for high-cardinality attributes. Scientific data, which is typically read (or append) only, have been shown to be better served with bitmap-based indexing techniques [20, 24], providing faster response times with lower index storage overhead.

While there are numerous technologies that use variants of bitmap indexing, we primarily focus on FASTBIT [24], a state-of-the-art bitmap indexing scheme, that is used by a number of scientific applications for answering range queries. FASTBIT employs a Word-Aligned-Hybrid (WAH) compression scheme based on run-length encoding, which decreases the index storage requirement and allows FASTBIT to perform logical operations efficiently on the compressed index and compute partial results by scanning the index. For those records that cannot be evaluated with the index alone, FASTBIT resorts to performing a read of the raw data, in what is called *candidate checks*. Unfortunately, the bitmap index created is sensitive to the distribution and cardinality of the input data, taking anywhere from 30 to 300% of the raw column size. The space can partly be reduced through techniques such as *precision binning*, at the cost of disturbing the distribution of values along the bins.

On the other side of the coin, data compression methods within databases have been widely studied [13, 16, 22]. For example, the column-oriented database C-Store [6] uses null compression (elimination of zeroes), dictionary encoding, and run-length encoding for effective data reduction of attributes organized contiguously, as opposed to the tra-

ditional row-store organization. MonetDB, on the other hand, uses the patched frame of reference (PFOR) algorithm and variants, which promotes extremely fast decompression speeds for query processing [27]. While these methods have limited use on double-precision data due to high-entropy significand bits, our work does share similarity with the dictionary encoding method, in that we compress floating-point data through identifying unique values and assigning them reduced bitwise representations. However, we perform this on only the most significant few bytes of the double-precision data, as opposed to the full dataset as in C-Store, and discard the representation entirely when using the inverted index for our query processing methodology.

As mentioned, many general-purpose and specialized compression methodologies fail to provide high compression ratios on double-precision data. Part of the reason for this is that floating-point scientific data is notoriously difficult to compress due to high entropy significands, of which floating-point data is primarily composed of (23 of 32 bits for single precision and 52 of 64 bits for double-precision). Much work has been done to build compressors for these kinds of data, mostly based on difference coding. Algorithms such as FPC [8] and fpzip [19] use *predictors* like the Lorenzo predictor [14], FCM [25] and DFCM [12] to compress. Given an input stream of double-precision values, the predictors use the previously seen values to predict the next value in the stream, and rather than attempt to compress the double values themselves, the compression algorithm uses a measure of error between the predicted and actual value, typically as an XOR operation.

Our methodology is based on treating the most significant bytes of double-precision data differently than the least significant bytes. Isenburg *et al.* use the same underlying concept in a prediction-based compression utility, which partitions the sign, exponent, and significand bits of the prediction error, followed by compression of each component [15]. Unlike their method, our method must maintain the approximability of the floating point datasets by treating the most significant bytes as a single component (sign, exponent, and the most significant significand bits), enabling efficient index generation and range query processing over the compressed data.

3 Method

3.1 System Overview

As mentioned, the goal of this paper is to facilitate query-driven analysis of large-scale scientific simulation data with storage-bound requirements. There are two stages where we focus our design to achieve this goal: first, while simulation data is being generated and in memory, or as a post-processing step, we can process and reorganize a double-precision dataset to compress the data. Second, we can modify the new organization of data to optimize query processing on the preprocessed data. For this purpose, we introduce two components in the scientific knowledge discovery pipeline as shown in Figure 1, referred to as the *lossless compressor* and *query engine*.

3.2 Compression

As mentioned, scientific simulations use predominantly double precision floating-point variables, so the remainder of the paper will focus on compression and query processing for these variables, though our method can be applied to variables of different precision.

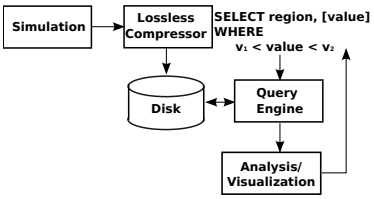


Fig. 1. System Workflow, Simulation to Analysis.

Our key observation for the compression process is that there is similarity with respect to orders of magnitude in our target datasets. For instance, in a simulation grid, adjacent grid values are unlikely to differ in orders of magnitude, except perhaps along simulation-specific phenomenon boundaries. Therefore, it is useful to consider utilizing the commonality in the encoding of the floating-point value. The IEEE 754 floating point standard encodes floating point values using a single *sign* bit and splits the remaining bits into the *significand* and the *exponent*. For 64-bit double precision values, this would be one sign bit, 11 exponent bits, and 52 significand bits. Given the sign bit s, the unsigned integral representation of the exponent field e, and each significand bit m_i (most to least significant), the resulting value encoded by a double-precision variable is:

$$\text{value} = (-1)^s \times (1 + \sum_{i=1}^{52} (m_i 2^{-i})) \times 2^{e-1023}. \quad (1)$$

Note that, all other components being equal, a difference of one in the exponent fields of two double-precision variables leads to a 2x difference in the represented values.

Hence, in the interest of compression efficiency, we want to split the (low-entropy) exponent and (possibly high-entropy) significand and compress separately. In the interest of computational efficiency, we want to use aligned byte boundaries. To this end, we split double precision datasets into two components: the k most significant bytes, or *high-order bytes*, and the remaining $(8-k)$ least significant bytes, or *low-order bytes*. We separate these for every point and build two data streams, one of which containing the k high-order bytes, and the other containing the remainder. In the high-order byte stream, we identify the distinct high-order bytes and discard the redundant values. In the low-order byte stream, we partition the column data into subsets, each corresponding to a distinct high-order byte pattern, mapping the record IDs (RIDs) appropriately. The value of k should be chosen with two goals in mind: to cause the cumulative number of distinct high-order bytes to stabilize with an increasing stream size, and to maximize the redundancy of the patterns (for compression) while encoding the entirety of the sign and exponent components (for future query processing).

However, with an eye toward utilizing the compression data for query processing, we handle the data in a slightly more complicated fashion. Figure 2 gives an overview of the compression system. We define a *partition* to be a compression stream of bounded maximum size. We define a *bin* to be a set of values with equivalent high-order bytes. Since we are working under the assumption of highly similar high-order bytes, we expect the number of bins B to be much smaller than the maximum number possible (for

$k = 2$, a maximum of 65,536). Therefore, we record the *unique* high-order bytes in the input data for each partition, forming the bins.

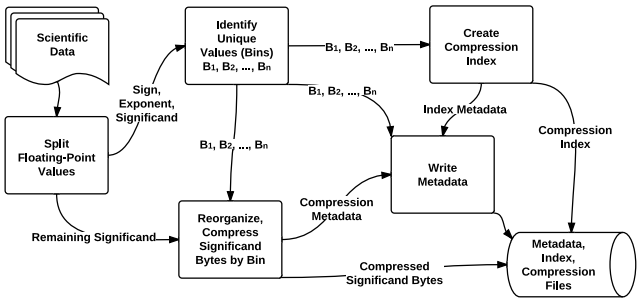


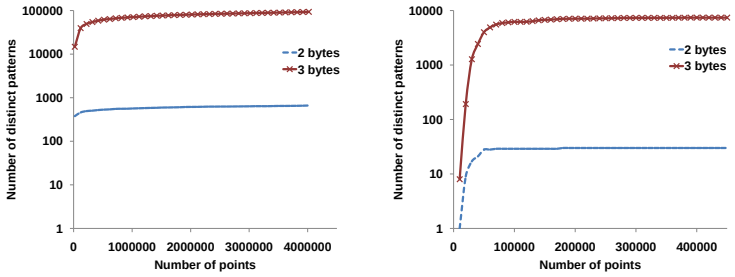
Fig. 2. Various stages of the compression methodology, described in Section 3.2. The bitmap index is used for compression, while the inverted index is used in query processing.

Once the bins are defined, we reorganize the low-order bytes to be contiguous in each bin, eliminating the need to map each low-order byte component to its respective high-order bytes. However, to maintain the record ID to bin mapping (when not optimizing for query processing), we build a bitmap that identifies each RID with a bin, using a size per identifier logarithmic to the number of bins. We then compress the bitmap and the low-order bytes using the general-purpose compressor bzip2 [1]. We do not consider using more complex algorithms, such as prediction-based compressors, in this paper. We feel that the use of a general-purpose compression algorithm provides a solid baseline of performance that applications can improve on, given additional application-specific knowledge of dataset characteristics.

Three data structures are produced as the result of the compression process: (1) the compression metadata, defining the high-order byte values and file offsets of each bin, (2) the compressed RID to bin index, mapping each index to the bin where the record's corresponding value is located, and (3) the bin-partitioned low-order bytes.

For scientific floating point data, we found $k = 2$ to be the most effective; it covers the sign bit, all exponent bits, and the first four significand bits of double precision values (approximately two significant figures in base 10 scientific notation). This makes sense, as higher degrees of precision in scientific data tend toward high-entropy values. To verify our choice of k for this paper, Figure 3 shows the number of distinct high-order bytes recorded as a data stream is processed. For both $k = 2$ and 3, a relatively small cardinality is seen relative to the number of points processed, with the distinct values quickly reaching a (near) maximum.

Recall that the metadata consists of unique high-order bytes as well as their respective file offsets to the low-order byte payload. Hence, the metadata size is directly proportional to the number of unique high-order bytes. As shown in Figure 3, for 2 of the scientific datasets, the size of metadata is less than 0.1% of the dataset for $k = 2$, due to the small number of distinct patterns. For $k = 3$, however, the number of distinct



(a) GTS potential dataset (b) S3D temperature dataset

Fig. 3. Cumulative growth of the number of distinct higher order 2-byte and 3-byte pattern for increasing data size.

patterns increases by a factor of 100 due to the addition of the higher-entropy significand bits. This increases the metadata size similarly, while additionally increasing the size of the RID to bin mapping logarithmically. Thus, we use $k = 2$ in this paper. Given the trends in Figure 3, we expect random sampling to be sufficient to determine a good value of k for double-precision datasets.

3.3 Query Processing: Index Generation

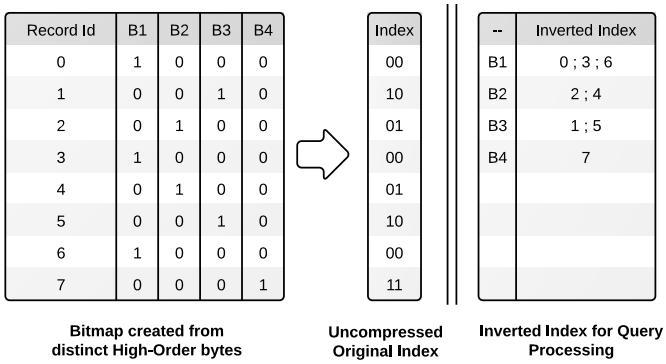


Fig. 4. Building an inverted index for query processing, compared to the index used in compression.

The compression methodology presented in Section 3.2 is, as will be shown, effective at improving the compression ratio of many scientific datasets, but is not optimized for query processing. If a range query is performed using our compression index, the entire bitmap would need to be traversed to map the binned data back to RIDs. Thus, at the cost of additional storage, we optimize for range queries by using an inverted index which maps each bin to a list of RIDs sharing the same high-order bytes, creating

a *bin-based value-to-RID* mapping. Figure 4 illustrates the index used in compression compared to the inverted index. This organization is advantageous for range query processing, because we now access the RIDs by bin, the same as accessing the low-order bytes. The organization is disadvantageous because of the increased space, both for the index itself as well as the additional metadata, such as file offsets, needed to access the new index. This means, for a partition of N elements, approximately $N\log(N)$ bits is needed to store the index, with marginally additional space to store metadata such as the number of elements within each bin. Bounding the maximum partition size to 32GB of double-precision data ensures that each RID in the inverted index needs no more than four bytes, making the index size less than 50% of the raw column size, or lower for smaller partitions. As a simple example, a partition size of 2GB of double-precision data requires 28 bits for each RID, translating to an index size of 43.75% of the raw column size. This is assuming, of course, that the partition is completely filled. Furthermore, we do not consider compression of the inverted indexes, a well-studied topic [23, 26] that we hope to integrate into our method and evaluate in the future.

3.4 Query Processing: File Layout

The data used by the query processing engine is split into three components: a metadata file, an index file, and a compression file, each corresponding to its purpose described in the previous sections. The metadata file is shown in Figure 5.

The metadata file contains partition information, including file offsets for each partition and bin, the number and bounds (high-order bytes) of bins, and the number of values per bin per partition. The index file and the compression file contain the RIDs and compressed low-order bytes, respectively. A single scan of the metadata file is necessary for query processing and is small enough to be held in memory to optimize future queries. In our experimentation, however, we do not consider this possibility.

<N number of partitions>
<Metadata offset for partition t> $(0 \leq t < N)$
<Index offset for partition t> $(0 \leq t < N)$
<Compression offset for partition t> $(0 \leq t < N)$
(Repeat for $0 \leq t < N$)
<M number of elements in partition t>
<B number of bins>
<Number of elements in bin b> $(0 \leq b < B)$
<Bin bound b> $(0 \leq b < B)$
<Compression offset b> $(0 \leq b < B)$
(End Repeat)

Fig. 5. Metadata file format.

3.5 Query Processing: Range Queries

Given a variable constraint $[v_1, v_2)$, a number of steps occur to retrieve the relevant values, based on a traversal of the metadata file shown in Figure 5. The process is shown in Figure 6. For each partition, relevant bins are identified using a method such

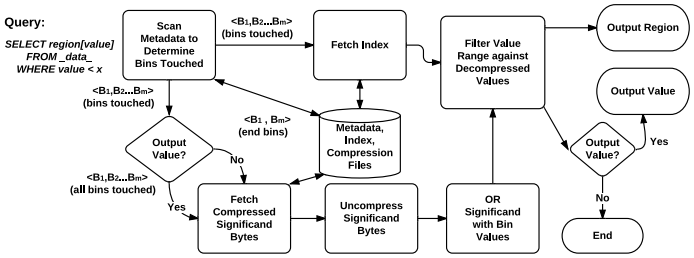


Fig. 6. Query processing methodology, taking into account metadata, index, and compression data fetching and aggregating.

as binary search, using the high-order bytes as a lower bound of each bin. Given a non-decreasing order of the bin organization in file, query processing per partition requires only a single seek in the index and the compression files to fetch the values associated with bins of interest. The low-order bytes and spatial indices are read, decompressed, and reconstructed into the original values, filtering out the ones that do not match the query.

In the case of queries requesting only RIDs, not all of the low-order bytes need to be fetched and reconstructed. Only the bins at each boundary need be checked for query bounds, as all remaining bins are guaranteed to fit within the query bounds.

4 Results And Discussions

4.1 Experimental Setup

We performed our experiments on the Lens cluster on Oak Ridge National Laboratory that is dedicated for high-end visualization and data analysis. Each node in the cluster is made up of four quad-core 2.3 GHz AMD Opteron processors and is equipped with 64GB of memory. In the following figures and tables, we refer to our methodology as CDI, corresponding to the Compressed representation of both the column Data and Index. All experiments were run with data located on the Lustre filesystem. For the indexing and query processing experiments, we compare against WAH encoding within the FASTBIT software. To avoid database-related overheads such as concurrency control, transaction support, etc. and provide a fair comparison between technologies, we wrote a minimal query driver for FASTBIT using only the necessary indexing and querying functions provided in the FASTBIT API. Furthermore, for fairness of comparison, we use the same partition size of 2GB for both our method and FASTBIT.

4.2 Query Processing

Index Generation

To evaluate the performance of our index generation methodology with respect to both computational efficiency as well as storage efficiency, we ran benchmarks on 13 scientific datasets, many of which are publicly available and discussed by Burtscher and Ratanaworabhan [9], and gathered throughput and storage-related metrics. Table 1 shows the results that we obtained from these experiments. For all 13 datasets, our indexing

strategy provides a throughput several times higher than that offered by FASTBIT [24]. On the storage front, the total storage footprint for our method, consisting of both index and data, is significantly less than FASTBIT with precision-3 binning on all of the 13 datasets, and less than FASTBIT with precision-2 binning on 11 of the 13 datasets. For the other two datasets (*msg_sp* and *num_brain*), we require slightly more space than FASTBIT with precision-2 binning, which we attribute to the variable's much lower cardinality relative to the other datasets. Our method's ability to provide high throughput in the index generation process makes it ideally suited for index building at simulation time, where minimizing simulation delay is a must.

Table 1. Query index generation throughput and storage footprint. FB-D: FASTBIT with default configuration (10^5 bins). FB-P2,3: FASTBIT with bin boundaries at two/three significant digits.

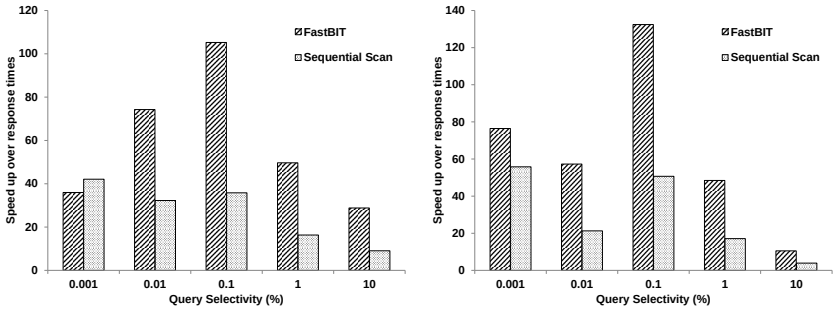
Dataset	**Index Gen. (MB/sec)**			**Storage (data+index) Req. (%)**			
	In-situ	**Post-processing**					
	CDI	CDI	FB-P3	CDI	FB-P2	FB-P3	FB-D
msg_bt	**166.75**	113.52	17.02	**119.36**	137.99	167.82	192.58
msg_lu	**474.92**	105.71	16.51	**124.44**	147.38	186.02	201.55
msg_sp	**481.85**	133.75	19.55	124.01	**115.62**	140.85	197.67
msg_sppm	**497.16**	148.63	25.46	**59.60**	109.80	114.9	125.32
msg_sweep3d	**510.10**	120.52	18.63	**96.62**	110.45	148.39	200.86
num_brain	**513.88**	88.03	19.51	124.50	**122.93**	165.53	202.31
num_comet	**404.26**	64.63	15.80	**116.20**	150.32	181.4	196.06
num_control	**370.23**	53.92	14.99	**124.05**	154.83	190.26	200.89
num_plasma	**503.99**	110.27	17.36	**51.44**	126.15	157.35	197.56
obs_error	**455.61**	90.00	11.47	**94.90**	130.34	157.85	176.93
obs_info	**498.35**	97.87	20.73	**75.06**	117.53	154.43	219.32
obs_spitzer	**478.20**	61.49	18.91	**94.37**	138.29	166.82	198.31
obs_temp	**397.18**	54.19	10.57	**125.03**	174.65	197.25	209.95

End-to-End Query Performance Evaluation

For an end-to-end performance comparison, we perform queries under a number of scenarios. We look at two types of range queries: those that output record IDs given constraints on variables, which we will refer to as "region-centric" queries, named for the use-case of retrieving "regions of interest" from datasets arranged on a spatial grid structure, and those that additionally output the values of the variables, which we will refer to as "value-centric" queries. We compare each of these query types against FASTBIT, which is specifically optimized for range queries.

We used following two scientific simulation datasets to evaluate our query performance in terms of value centric queries and region centric queries.

1. GTS [21]: A particle-based simulation for studying plasma microturbulence in the core of magnetically confined fusion plasmas of toroidal devices.
2. S3D [10]: A first-principles-based direct numerical simulation (DNS) of reacting flows which aids the modeling and design of combustion devices.



(a) GTS Potential data (b) S3D Temperature data

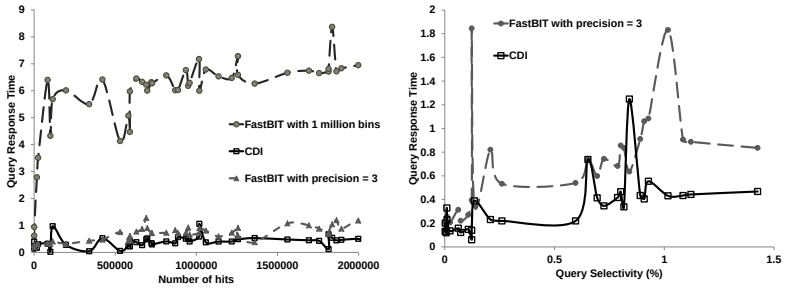
Fig. 7. Comparison of speedup of our method over FASTBIT and sequential scans, for value-centric queries when the query selectivity is varied from 0.001% to 10.0%.

Value-centric Queries Figure 7 shows the speedup in value-based query response time using our method, compared to FASTBIT's default and precision-based indexing, with varying query selectivity. By query selectivity, we refer to the percentage of the raw dataset returned by a query. For two scientific application variables S3D velocity and GTS potential, we provide a speedup of greater than a factor of 28. Due to the clustering of the data, a very small number of I/O seek operations are needed by our method as opposed to FASTBIT. The reason that sequential scan performs better than FASTBIT in this context is that, in parallel file systems such as Lustre, seeks are a very high-latency operation; FASTBIT resorts to seeking per item, while sequential scan reads all data in a single read.

For value-centric queries, not much difference is observed in the response time by FASTBIT using precision binning and default binning. This is because, in both cases, random disk access dominates processing time. While FASTBIT has a very fast CPU processing time for each query, the I/O time spent on random file access dominates the overall query response time.

The speedup observed increases from a factor of 35 for 0.001% selectivity to 105 for 0.1% selectivity. Here the performance improvement is due to a significantly lower number of seeks. On increasing query selectivity, FASTBIT can fetch more consecutive blocks of file from disk, thus reducing I/O seek time. The I/O read time contributes to most of the query response time. Thus, the speedup comes down for 10% selectivity to a factor of about 28.

Region-centric Queries Figure 8 shows region query response time with varying number of hits (records returned) for our method compared to FASTBIT with precision and default binning. For region-centric queries, only the points falling within misaligned bins need to be evaluated. For FASTBIT, the type of binning used plays a definitive role in determining the time taken to respond to region queries. In the case of precision binning for FASTBIT, it can answer queries involving three decimal point precision by going through the bitmap alone. It need not seek to the disk if the range specified in the



(a) GTS Potential data (b) S3D Temperature data

Fig. 8. Comparison of response return by FASTBIT against our method for region-centric queries with varying number of query hits.

query involves less than three decimal points. On the other hand, the default binning option needs to perform raw data access to evaluate edge bins.

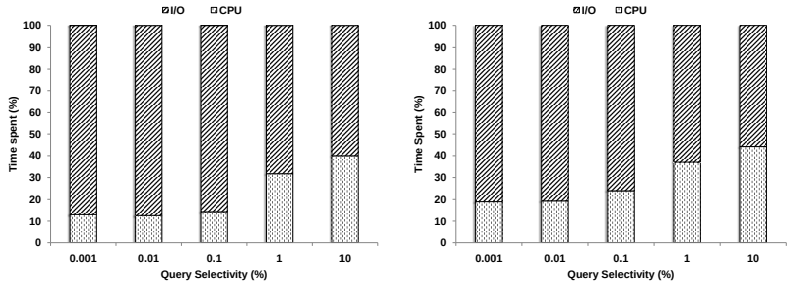
The performance of our method is better than the precision binning in many cases, but both methods see instances of lower performance. This is caused by partitioning methods that split on a fixed, rather than arbitrary, precision, causing lower degree of regularity between the bins. This happens when misaligned bins happen to be those with the largest number of points contained in them. In these cases, there is a higher false positive rate, causing it to be slower than FASTBIT, though FASTBIT is seen to have similar issues when using the precision-binning option.

4.3 Performance Analysis

Figure 9 shows the breakup of overall processing time into I/O and compute components, corresponding to index/bin loading and decompression, respectively. The dataset tested on is S3D using the velocity variable. I/O is still a significant portion of the time, but much of the overall time is spent decompressing the data. Combined with the superior performance in many cases compared to FASTBIT, we believe that the portion spent on decompressing is a positive aspect of our system, as it takes much of the pressure off of I/O operations, which is certainly good for performance in HPC contexts. Furthermore, this opens up windows for performing clever optimizations, such as multithreading I/O and decompression to hide the compute time, improving performance to an even higher degree than already seen.

4.4 Compression

To analyze the performance of our lossless data compression scheme, we compared the compression ratios obtained with our method (without the inverted index) to those obtained by other standard lossless compression utilities on a total of 26 scientific datasets, including application datasets such as Flash [7], GTS [21], S3D [10], XGC [18]. Out of these datasets, our method performed better than all of the other compressors tested (gzip [2], fpzip [19], bzip2 [1], and FPC [9]) on 21 of 27. FPC gave superior performance compared to our method on two of the 27 datasets, *msg_sweep3d* and *obs_error*, fpzip gave better performance on three of the 27, the *flash* datasets, and bzip2 gave bet-



(a) Time distribution for value-centric queries

(b) Time distribution for region-centric queries

Fig. 9. Comparison of computation and I/O time distribution for our method for different query types over varying selectivity. On the other hand, FASTBIT spends over 90% of the time on I/O.

ter performance on only one of the 27, *xgc_igid*. Overall, our method was consistent in yielding comparable or better compression ratios than the other compressors, providing evidence of strong compression ratios in other application datasets.

To substantiate our superior performance on most of the datasets, we argue that the bin-based compression of the data generally allows a much greater exploitation of existing compression algorithms than the normal distribution of scientific data that was passed to the other compressors. The reorganization of the data allowed gzip and bzip2's algorithms to be utilized as best as possible, causing the data to be reduced significantly because of the splitting of the low-entropy and high-entropy sections of the data. In regards to the datasets that FPC performed better than our method on, we attribute the performance difference to the encoding of data dependency which the FCM [25], DFCM [12], and Lorenzo [14] predictors used by FPC were able to capture in their predictions. In regards to bzip2's superior performance on the *xgc_igid* dataset, the reorganization of already predictable data along with the overheads caused by our method likely caused the loss in compression improvement.

5 Conclusion

As the size of scientific datasets in various disciplines continues to grow, new methods to store and analyze the datasets must be developed, as I/O capabilities are not growing as fast, and new technologies, such as SSDs are not currently able to achieve the storage density and cost-efficiency of traditional mechanical disk drives. Successful methods of mitigating this growing gap must involve data reduction in all stages of the knowledge discovery pipeline, including storage of raw data as well as analytics metadata. We believe our effort at compression, indexing, and query processing of scientific data represents a step in the right direction, allowing both efficient lossless compression of double-precision data for accuracy-sensitive applications as well as efficient query processing on variable constraints, all with less space and I/O requirements than other database technologies.

Table 2. Compression ratio and data storage requirement for different components.

Dataset	Compression Ratio					Storage Requirement (%)		
	gzip	fpzip	bzip2	FPC	CDI	Data	Index	Metadata
msg_bt	1.12	1.20	1.09	1.29	**1.40**	69.35	1.86	≈0.00
msg_lu	1.05	1.13	1.01	1.17	**1.30**	74.42	1.97	0.01
msg_sp	1.10	1.11	1.06	1.26	**1.33**	73.98	1.11	≈0.00
msg_sppm	7.41	3.25	7.09	5.30	**8.87**	9.58	1.66	0.02
msg_sweep3d	1.09	1.33	1.32	**3.09**	2.11	46.60	0.67	0.02
num_brain	1.06	1.25	1.06	1.16	**1.28**	74.50	3.39	≈0.00
num_comet	1.16	1.27	1.17	1.16	**1.34**	66.16	8.16	0.03
num_control	1.05	1.12	1.03	1.05	**1.15**	74.02	12.22	0.02
num_plasma	1.77	1.06	6.17	15.05	**80.67**	1.40	1.04	0.03
obs_error	1.44	1.37	1.36	**3.60**	2.59	44.90	5.88	≈0.00
obs_info	1.14	1.06	1.22	2.27	**3.52**	24.97	3.36	0.04
obs_spitzer	1.23	1.07	1.78	1.03	**1.90**	44.36	8.05	≈0.00
obs_temp	1.03	1.09	1.03	1.02	**1.13**	75.00	12.70	0.03
gts_phi_l	1.04	1.18	1.02	1.07	**1.19**	75.00	8.56	0.03
gts_phi_nl	1.04	1.17	1.01	1.07	**1.18**	75.00	9.2	0.03
gts_chkp_zeon	1.04	1.09	1.02	1.01	**1.17**	75.00	10.04	0.10
gts_chkp_zion	1.04	1.10	1.02	1.02	**1.18**	75.00	9.6	0.11
gts_potential	1.04	1.15	1.01	1.06	**1.18**	75.00	9.60	≈0.00
xgc_igid	3.00	2.73	**3.12**	1.96	3.05	32.74	0	≈0.00
xgc_iphase	1.36	1.53	1.37	1.36	**1.58**	55.33	7.56	0.00
xgc_restart	1.45	1.26	1.45	1.35	**1.67**	52.84	6.72	≈0.00
s3d_temp	1.33	1.94	1.45	1.23	**1.89**	48.16	4.68	0.00
s3d_vmag	1.19	1.71	1.21	1.16	**1.77**	50.00	4.31	0.01
s3d_vvel	1.04	1.24	1.02	1.15	**1.27**	75.00	3.74	≈0.00
flash_velx	1.11	**1.34**	1.08	1.26	1.31	75.00	0.81	≈0.00
flash_vely	1.13	**1.43**	1.09	1.29	1.31	75.00	0.80	≈0.00
flash_gamc	1.28	**1.62**	1.28	1.53	1.40	71.37	0.06	≈0.00

References

1. bzip2, 2007. `http://www.bzip.org/`.
2. GNU gzip, 2007. `http://www.gzip.org/`.
3. MySQL, 2010. `http://www.mysql.com/`.
4. Oracle, 2010. `http://www.oracle.com/`.
5. IBM DB2, 2011. `http://www-01.ibm.com/software/data/db2/`.
6. D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.
7. B. Fryxell and K. Olson and P. Ricker and F. X. Timmes and M. Zingale and D. Q. Lamb and P. MacNeice and R. Rosner and J. W. Truran and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, Nov 2000.
8. M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *IEEE Data Compression Conference*, pages 293–302, 2007.

9. M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58:18–31, 2009.
10. J. H. Chen, A. Choudhary, B. Supinski, M. DeVries, S. K. E.R. Hawkes, W. Liao, K. Ma, J. Mellor-Crummey, N. Podhorszki, S. S. R. Sankaran, and C. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Comp. Sci. and Discovery*, 2(1).
11. D. Comer. The ubiquitous B-Tree. *ACM Comput. Surv.*, 11:121–137, June 1979.
12. B. Goeman, H. Vandierendonck, and K. D. Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Seventh International Symposium on High Performance Computer Architecture*, pages 207–216, 2001.
13. G. Graefe and L. Shapiro. Data compression and database performance. In *Applied Computing, 1991., [Proceedings of the 1991] Symposium on*, pages 22 –27, apr 1991.
14. L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22:343–348, 2003.
15. M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless compression of predicted floating-point geometry. *JCAD - Journal for Computer-Aided Design*, 37:2005, 2004.
16. B. R. Iyer and D. Wilhite. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 695–704, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
17. K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rubel, A. Shoshani, A. Sim, K. Stockinger, G. Weber and W-M Zhang. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.
18. S. Ku, C. Chang, and P. Diamond. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic Tokamak geometry. *Nuclear Fusion*, 49(11):115021, 2009.
19. P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12:1245–1250, 2006.
20. R. R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32, August 2007.
21. W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas*, 13(9):092505, 2006.
22. T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, Sept. 2000.
23. I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.
24. K. Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series*, volume 16, page 556. IOP Publishing, 2005.
25. S. Yiannakis and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.
26. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.
27. M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.